DJANGO

What is Django ?

Django is a high-level Python web framework that enables rapid development of secure and maintainable websites. Built by experienced developers, Django takes care of much of the hassle of web development, so you can focus on writing your app without needing to reinvent the wheel. It is free and open source, has a thriving and active community, great documentation, and many options for free and paid-for support.

Django helps you write software that is:

- **Complete**

  Django follows the "Batteries included" philosophy and provides almost everything developers might want to do "out of the box". Because everything you need is part of the one "product", it all works seamlessly together, follows consistent design principles, and has extensive.

- **Versatile**

  Django can be (and has been) used to build almost any type of website — from content management systems and wikis, through to social networks and news sites. It can work with any client-side framework, and can deliver content in almost any format (including HTML, RSS feeds, JSON, XML, etc).

  Internally, while it provides choices for almost any functionality you might want (e.g. several popular databases, templating engines, etc.), it can also be extended to use other components if needed.

- **Secure**

  Django helps developers avoid many common security mistakes by providing a framework that has been engineered to "do the right things" to protect the website automatically. For example, Django provides a secure way to manage user accounts and passwords, avoiding common mistakes like putting session information in cookies where it is vulnerable (instead cookies just contain a key, and the actual data is stored in the database) or directly storing passwords rather than a password hash.

  A password hash is a fixed-length value created by sending the password through a [cryptographic hash function](#) *Django can check if an entered password is correct by running it through the hash function and comparing the output to the stored hash value. However due to the "one-way" nature of the function, even if a stored hash value is compromised it is hard for an attacker to work out the original password.*

  Django enables protection against many vulnerabilities by default, including SQL injection, cross-site scripting, cross-site request forgery and clickjacking

- **Scalable**

  Django uses a component-based "[shared-nothing](#)" architecture (each part of the architecture is independent of the others, and can hence be replaced or changed if needed). Having a clear separation between the different parts means that it can scale for increased traffic by adding hardware at any level: caching servers, database servers, or application servers. Some of the busiest sites have successfully scaled Django to meet their demands (e.g. Instagram and Disqus, to name just two).

- **Maintainable**

  Django code is written using design principles and patterns that encourage the creation of maintainable and reusable code. In particular, it makes use of the Don't Repeat Yourself (DRY) principle so there is no unnecessary duplication, reducing the amount of code. Django also promotes the grouping of related functionality into reusable "applications" and, at a lower level, groups related code into modules.

- **Portable**

  Django is written in Python, which runs on many platforms. That means that you are not tied to any particular server platform, and can run your applications on many flavours of Linux, Windows, and Mac OS X. Furthermore, Django is well-supported by many web hosting providers, who often provide specific infrastructure and documentation for hosting Django sites.

Where did it come from?

Django was initially developed between 2003 and 2005 by a web team who were responsible for creating and maintaining newspaper websites. After creating a number of sites, the team began to factor out and reuse lots of common code and design patterns. This common code evolved into a generic web development framework, which was open-sourced as the "Django" project in July 2005.

Is Django opinionated?

Web frameworks often refer to themselves as "opinionated" or "unopinionated".

Opinionated frameworks are those with opinions about the "right way" to handle any particular task. They often support rapid development *in a particular domain* (solving problems of a particular type) because the right way to do anything is usually well-understood and well-documented. However they can be less flexible at solving problems outside their main domain, and tend to offer fewer choices for what components and approaches they can use.
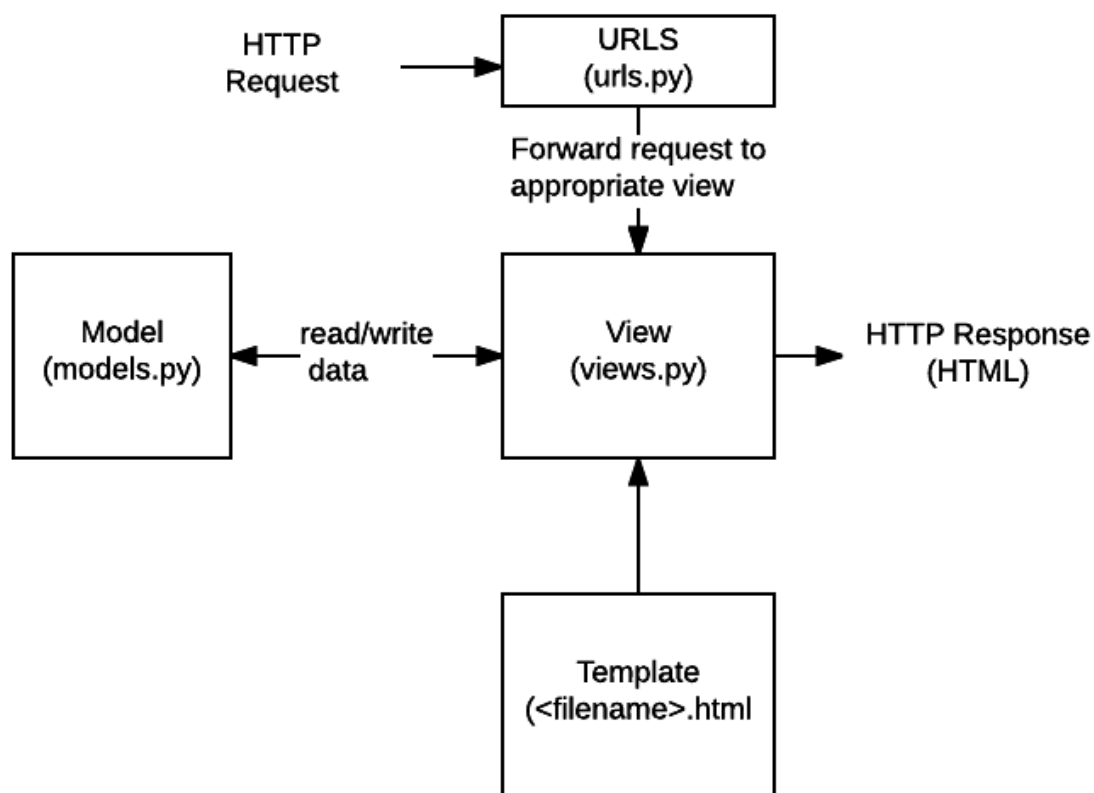
Unopinionated frameworks, by contrast, have far fewer restrictions on the best way to glue components together to achieve a goal, or even what components should be used. They make it easier for developers to use the most suitable tools to complete a particular task, albeit at the cost that you need to find those components yourself.

Django is "somewhat opinionated", and hence delivers the "best of both worlds". It provides a set of components to handle most web development tasks and one (or two) preferred ways to use them. However, Django's decoupled architecture means that you can usually pick and choose from a number of different options, or add support for completely new ones if desired.

What does Django code look like?

In a traditional data-driven website, a web application waits for HTTP requests from the web browser (or other client). When a request is received the application works out what is needed based on the URL and possibly information in POST data or GET data. Depending on what is required it may then read or write information from a database or perform other tasks required to satisfy the request. The application will then return a response to the web browser, often dynamically creating an HTML page for the browser to display by inserting the retrieved data into placeholders in an HTML template.

Django web applications typically group the code that handles each of these steps into separate files:

- **URLs:**

    While it is possible to process requests from every single URL via a single function, it is much more maintainable to write a separate view function to handle each resource. A URL mapper is used to redirect HTTP requests to the appropriate view based on the request URL. The URL mapper can also match particular patterns of strings or digits that appear in a URL and pass these to a view function as data.

    - **Sending the request to the right view(urls.py)**

        A URL mapper is typically stored in a file named urls.py. **In the example below, the mapper (urlpatterns) defines a list of mappings between** *routes* **(specific URL** *patterns)* **and corresponding view functions.**

        ```
        urlpatterns = [
            path('admin/', admin.site.urls),
            path('book/<int:id>/', views.book_detail, name='book_detail'),
            path('catalog/', include('catalog.urls')),
            re_path(r'^([0-9]+)/$', views.best),
        ]
        ```

        The **urlpatterns** object is a list of **path()** and/or **re_path()** functions.
        The **path()** method uses angle brackets to define parts of a URL that will be captured and passed through to the view function as named arguments.
        The **re_path()** function uses a flexible pattern matching approach known as a regular expression. We'll talk about these in a later article!

- **View**

    A view is a request handler function, which receives HTTP requests and returns HTTP responses. Views access the data needed to satisfy requests via *models,* and delegate the formatting of the response to *templates.*

    - **Handling the request(views.py)**

        Views are the heart of the web application, receiving HTTP requests from web clients and returning HTTP responses.
        The example below shows a minimal view function **index(),** which could have been called by our URL mapper in the previous section. Like all view functions it receives an **HttpRequest** object as a parameter **(request)** and returns an **HttpResponse** object.

        ```
        # filename: views.py (Django view functions)

        from django.http import HttpResponse

        def index(request):
            # Get an HttpRequest - the request parameter
            # perform operations using information from the request.
            # Return HttpResponse
            return HttpResponse('Hello from Django!')
        ```

o **Querying the request (views.py)**
The Django model provides a simple query API for searching the associated database. This can match against a number of fields at a time using different criteria (e.g. exact, case-insensitive, greater than, etc.), and can support complex statements The code snippet shows a view function (resource handler) for displaying all of our U09 teams.

```python
## filename: views.py

from django.shortcuts import render
from .models import Team

def index(request):
    list_teams = Team.objects.filter(team_level__exact="U09")
    context = {'youngest_teams': list_teams}
    return render(request, '/best/index.html', context)
```

- **Models**
Models are Python objects that define the structure of an application's data, and provide mechanisms to manage (add, modify, delete) and query records in the database.
  o **Defining data models (models.py)**
  Django web applications manage and query data through Python objects referred to as models. Models define the structure of stored data, including the field *types* and possibly also their maximum size, default values, selection list options, help text for documentation, label text for forms, etc.
  The code snippet below shows a very simple Django model for a Team object.

```python
# filename: models.py

from django.db import models

class Team(models.Model):
    team_name = models.CharField(max_length=40)

    TEAM_LEVELS = (
        ('U09', 'Under 09s'),
        ('U10', 'Under 10s'),
        ('U11', 'Under 11s'),
        ...  #list other team levels
    )
    team_level = models.CharField(max_length=3, choices=TEAM_LEVELS, default='U11')
```

- **Templates**
  A template is a text file defining the structure or layout of a file (such as an HTML page), with placeholders used to represent actual content. A *view* can dynamically create an HTML page using an HTML template, populating it with data from a *model*. A template can be used to define the structure of any type of file; it doesn't have to be HTML!
  - **Rendering data (templates.py)**
    Template systems allow you to specify the structure of an output document, using placeholders for data that will be filled in when a page is generated. Templates are often used to create HTML, but can also create other types of document.

```html
## filename: best/templates/best/index.html

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Home page</title>
</head>
<body>
  {% if youngest_teams %}
    <ul>
      {% for team in youngest_teams %}
        <li>{{ team.team_name }}</li>
      {% endfor %}
    </ul>
  {% else %}
    <p>No teams are available.</p>
  {% endif %}
</body>
</html>
```

The preceding sections show the main features that you'll use in almost every web application: URL mapping, views, models and templates.

Just a few of the other things provided by Django include:

- **Forms**
  HTML Forms are used to collect user data for processing on the server. Django simplifies form creation, validation, and processing.
- **User authentication and permissions**
  Django includes a robust user authentication and permission system that has been built with security in mind.
- **Caching**
  Creating content dynamically is much more computationally intensive (and slow) than serving static content. Django provides flexible caching so that you can store all or part of a rendered page so that it doesn't get re-rendered except when necessary.
- **Administration site**
  The Django administration site is included by default when you create an app using the basic skeleton. It makes it trivially easy to provide an admin page for site administrators to create, edit, and view any data models in your site.
- **Serialising data**
  Django makes it easy to serialise and serve your data as XML or JSON. This can be useful when creating a web service (a website that purely serves data to be consumed by other applications or sites, and doesn't display anything itself), or when creating a website in which the client-side code handles all the rendering of data.